Reconciling High Server Utilization and Sub-millisecond Quality-of-Service

Jacob Leverich Christos Kozyrakis
Computer Science Department, Stanford University
{leverich,christos}@cs.stanford.edu

Abstract

The simplest strategy to guarantee good quality of service (QoS) for a latency-sensitive workload with sub-millisecond latency in a shared cluster environment is to never run other workloads concurrently with it on the same server. Unfortunately, this inevitably leads to low server utilization, reducing both the capability and cost effectiveness of the cluster.

In this paper, we analyze the challenges of maintaining high QoS for low-latency workloads when sharing servers with other workloads. We show that workload co-location leads to QoS violations due to increases in queuing delay, scheduling delay, and thread load imbalance. We present techniques that address these vulnerabilities, ranging from provisioning the latency-critical service in an interference aware manner, to replacing the Linux CFS scheduler with a scheduler that provides good latency guarantees and fairness for co-located workloads. Ultimately, we demonstrate that some latency-critical workloads can be aggressively co-located with other workloads, achieve good QoS, and that such co-location can improve a datacenter's effective throughput per TCO-$ by up to 52%.

1. Introduction

Warehouse-scale datacenters host tens of thousands of servers and consume tens of megawatts of power [14]. These facilities support popular online services such as search, social networking, webmail, video streaming, online maps, automatic translation, software as a service, and cloud computing platforms. We have come to expect that these services provide us with instantaneous, personalized, and contextual access to terabytes of data. Our high expectations of these services are largely due to the rapid rate of improvement in the capability (performance) and total cost of ownership (TCO) of the datacenters that host them.

Many factors that led to TCO and capability improvements are reaching the point of diminishing returns. Cost was initially reduced by switching from high-end servers to commodity x86 hardware and by eliminating the overheads of power distribution and cooling (PUE has dropped from 3.0 to 1.1) [14]. Unfortunately, these are both one-time improvements. In the past, we could also rely on deploying new servers with processors offering higher performance at the same power consumption; in effect, by replacing servers year after year, we could achieve greater compute capability without having to invest in new power or cooling infrastructure. However, the end of voltage scaling has resulted in a significant slowdown in processor performance scaling [9]. A datacenter operator can still increase capability by building more datacenters, but this comes at the cost of hundreds of millions of dollars per facility and is fraught with environmental, regulatory, and economic concerns.

These challenges have led researchers to pay attention to the utilization of existing datacenter resources. Various analyses estimate industry-wide utilization between 6% [16] and 12% [10, 40]. A recent study estimated server utilization on Amazon EC2 in the 3% to 17% range [21]. Even for operators that utilize advanced cluster management frameworks that multiplex workloads on the available resources [13, 32, 37], utilization is quite low. Hence, an obvious path towards improving both the capability and cost of datacenters is to make use of underutilized resources; in effect, raise utilization [6, 22, 23].

High utilization is straight-forward to achieve if system throughput is the only performance constraint: we can co-locate multiple workloads on each server in order to saturate resources, switching between them in a coarse-grain manner to amortize overheads. In contrast, high utilization is difficult to achieve in the presence of complex, latency-critical workloads such as user-facing services like search, social networking, or automatic translation [2]. For instance, updating a social networking news feed involves queries for the user's connections and his/her recent status updates; ranking, filtering, and formatting these updates; retrieving related media files; selecting and formatting relevant advertisements and recommendations; etc. Since multiple tiers and tens of servers are involved in each user query, low average latency from each server is not sufficient. These services require low tail latency (e.g., low 95th or 99th percentile) so that latency outliers do not impact end-to-end latency for the user [4].

The conventional wisdom is that latency-critical services do not perform well under co-location. The additional workloads can interfere with resources such as processing cores, cache space, memory or I/O bandwidth, in a manner that introduces high variability in latency and violates each ser-

EuroSys '14, April 13–16, 2014, Amsterdam, Netherlands.

ACM 978-1-4503-2704-6/14/04…$15.00.
http://dx.doi.org/10.1145/2592798.2592821

vice's QoS constraints. This concern drives operators to deploy latency-sensitive services on dedicated servers, or to grossly exaggerate their resource reservations on shared clusters. For instance, the Google cluster studied by Reiss showed average CPU and memory reservation of 75% and 60% of available resources, respectively, while actual utilization was only 20% and 40%.

The goal of this work is to investigate if workload co-location and good quality-of-service for latency-critical services are fundamentally incompatible in modern systems, or if instead we can reconcile the two. Using memcached, a widely deployed distributed caching service, as a representative workload with aggressive QoS requirements (hundreds of microseconds in many commercial deployments), we study the challenges and opportunities in co-locating latency-critical services with other workloads on the same servers. First, we analyze three common sources of QoS degradation due to co-location: (1) increases in queuing delay due to interference on shared resources (e.g., caches or memory), (2) long scheduling delays when timesharing processor cores, and (3) poor tail latency due to thread load imbalance. Second, we propose techniques and best practices to address these problems in existing servers. To manage queuing delay due to shared resource interference, we suggest interference-aware provisioning of the latency-critical service, so that interference leads to predictable decrease in throughput instead of intolerable spikes in latency. To address long scheduling delays, we find that Linux's CFS scheduler is unable to provide good latency guarantees while maintaining fairness between tasks. We demonstrate that a modern version of the BVT scheduler [7] affords predictable latency and fair sharing of CPU amongst tasks, and allows for aggressive co-location of latency-critical services. Finally, we find that thread-pinning and interrupt routing based on network flow-affinity resolve much of memcached's vulnerability to load imbalance.

We put the observations above together to evaluate the potential savings from co-locating latency-critical services with other workloads under two important scenarios. For example, with a memcached cluster utilized at 30%, co-location of batch workloads gives you the equivalent throughput of a 47% larger cluster that would otherwise cost 29% more. On a general-purpose cluster with 50% CPU utilization, we can co-locate memcached to harness stranded memory, and achieve memcached performance that would otherwise require a 17% increase in TCO. Overall, we show that high server utilization and strong QoS guarantees for latency-critical services are not incompatible, and that co-location of low-latency services with other workloads can be highly beneficial in modern datacenters.

2. Improving Utilization through Co-location

There are several reasons for the low utilization observed in datacenters today. Many companies have thousands of servers dedicated to latency-critical tasks, such as web-serving and key-value stores for user-data. These servers are under-utilized during periods of low traffic, such as evenings or weekends. Even worse, the number of deployed servers is typically determined by the requirements of traffic spikes during uncommon events (e.g., Black Friday, celebrity mishaps), so these servers are often under-utilized, even during periods of high nominal traffic. This creates an opportunity to use the spare capacity for other workloads. For instance, a company like Facebook, which has thousands of dedicated servers for memcached, could use spare CPU capacity to run analytics jobs that would otherwise run on a separate set of machines. This co-location scenario can be quite beneficial in terms of computing capability and cost (discussed in Sec. 5) as long as the analytics jobs do not unreasonably impact the quality-of-service of memcached. Such a system might be managed by a cluster scheduler that can adjust the number of analytics jobs as the load on memcached varies throughout the day [6, 41].

Underutilization also occurs because of the difficulty in allocating the right number of resources for jobs of any kind, whether latency-critical or throughput-oriented. Similarly, it is difficult to build servers that have the perfect balance of resources (processors, memory, etc.) for every single workload. The analysis of a 12,000-server Google cluster by Reiss et al. [32] shows that while the average reservation of CPU and memory resources is 75% and 60% of available resources respectively, the actual utilization is 20% and 40% respectively. One could deploy memcached as an additional service that uses the underutilized processing cores to export the underutilized memory on such a cluster for other uses, such as a distributed disk cache. For a 12,000-server cluster, this could provide a DRAM cache with capacity on the order of 0.5 petabytes at essentially no extra cost, provided the original workload is not disturbed and that memcached can serve this memory with good QoS guarantees.

One can imagine several additional scenarios where co-locating latency-critical services with other workloads will lead to significant improvements in datacenter capability and TCO. Nevertheless, most datacenters remain underutilized due to concerns about QoS for latency-critical services. We believe this is why most companies deploy latency-critical services, like memcached, on dedicated servers, and why latency-critical services running on shared machines have exaggerated resource reservations (as seen from Reiss' analysis of Google's cluster). Several recent works aim to discover when latency-critical services and other workloads do not interfere with each other, in order to determine when co-location is permissible [6, 23, 45]. However, this approach is pessimistic; it dances around the symptoms of interference but does not address the root causes. Our work provides a deeper analysis of the causes of QoS problems, which allows us to propose more effective mechanisms to raise utilization through co-location without impacting QoS.

3. Analysis of QoS Vulnerabilities

In this section, we perform an analysis of the QoS vulnerabilities of latency-critical services when co-located with other workloads. We focus on interference local to a server (processor cores, caches, memory, I/O devices and network adapters); QoS features for datacenter networks are studied elsewhere and not considered in this work [15].

The experimental portion of this analysis focuses on memcached. We chose memcached for several reasons. First, it has exceptionally low nominal latency, as low as any other widely deployed distributed service found in datacenters today (excluding certain financial "high-frequency trading" workloads). As such, it is quite sensitive to interference from co-located work, making it easy to identify and analyze their causes. Second, it is a concise, generic example of an event-based service, and many other widely deployed services (including REDIS, node.js, lighttpd, nginx, etc.) share many aspects of its basic architecture and use the same basic kernel mechanisms (epoll via libevent). Indeed, only 17% of CPU time when running memcached is spent in user-code, versus 83% in the kernel, so the particulars of memcached are less important than how it uses kernel mechanisms. Thus, we believe that a thorough analysis of memcached yields knowledge applicable to other services as well. Third, memcached consists of less than 10,000 lines of code, so it is easy to deconstruct. Finally, memcached is an important service in its own right, a cornerstone of several large web applications, and deployed on many thousands of servers. Knowledge gleaned from its careful analysis is useful, even outside the context of understanding interference from co-located work.

3.1 Analysis Methodology

There are a couple of prerequisite steps to our analysis. First, it is important to set realistic expectations about guarantees for latency under realistic load. For memcached, there is sufficient information about traffic patterns in large-scale commercial deployments (key and data sizes, put/get distributions, etc) in order to recreate realistic loads [1]. These deployments typically monitor tail latency and set QoS requirements thereof, such as requiring 95th-percentile latency to be lower than 500 to 1000 microseconds under heavy load. Second, it is equally important to carefully measure the single-request latency on an unloaded server. This measurement helps in multiple ways. First, if there is significant variability, it will be difficult to meet an aggressive QoS target, even without interference from other workloads, unless requests with high and low service times are prioritized and treated separately within the application. Second, the behavior of a well-tuned server can be approximated with a basic M/M/n queuing model (independent arrival and service time distributions), where n is the number of worker threads concurrently running on the server [11, 18, 24]. The unloaded latency (service time) allows us to estimate the performance potential of a well-tuned system and pinpoint when observed behavior deviates due to interference or any other reason.

The main part of our analysis consists of separating the impact of interference from co-located workloads into its three key causes: (1) *queuing delay*, (2) *scheduling delay*, and (3) *load imbalance*.

Queuing delay occurs due to coincident or rapid request arrivals. Even without co-location, the M/M/1 model suggests that, given a mean service rate of μ and mean arrival rate of λ, the average waiting time is $1/(\mu - \lambda)$ and 95th-percentile latency roughly $3\times$ higher ($ln(\frac{100}{100-95})/(\mu - \lambda)$). Interference from co-located workloads impacts queuing delay by increasing service time, thus decreasing service rate. Even if the co-located workload runs on separate processor cores, its footprint on shared caches, memory channels, and I/O channels slows down the service rate for the latency-critical workload. Even if a single request is only marginally slower in absolute terms, the slowdown is compounded over all queued requests and can cause a significant QoS problem. As λ approaches μ (i.e., at high load), wait time asymptotically increases, experienced as significant request latency by clients. It is common practice to take this asymptotic response into account when provisioning servers for a latency-sensitive service in order to ensure that no server is ever subjected to load that leads to excessive latency (e.g., provision servers for a load of at most 70-80% of μ). The impact of co-location on queuing delay can be thoroughly characterized by running the latency-critical service concurrently with micro-benchmarks that put increasing amounts of pressure on individual resources [5].

Distinct from queuing delay, scheduling delay becomes a QoS vulnerability when a co-located workload contends for processor cores with a latency-critical workload. That is, if the OS or system administrator assigns two tasks to the same core, they become subject to the scheduling decisions of the OS, which may induce long-enough delays to cause QoS violations. There are two parts to scheduling delay that must be considered independently: scheduler wait time, which is the duration of time that the OS forces a task to wait until it gets access to the core while another task runs, and context switch latency, which is the time it takes the OS and hardware to switch from running one application to another after the scheduling algorithm has determined that it is the others' turn to run. The impact of co-location on scheduling delay can be characterized by constructing co-location scenarios that exacerbate the priorities and idiosyncrasies of the scheduling algorithm. We present such scenarios for commonly used Linux schedulers in Sec. 3.2.5.

Finally, load imbalance across threads of a multi-threaded service can lead to poor overall tail latency, even when the cumulative load on a server is low and average or median latency appears nominal. Co-located work can exacerbate load imbalance in a multi-threaded service in numerous ways: by causing additional queuing delay on only the threads that

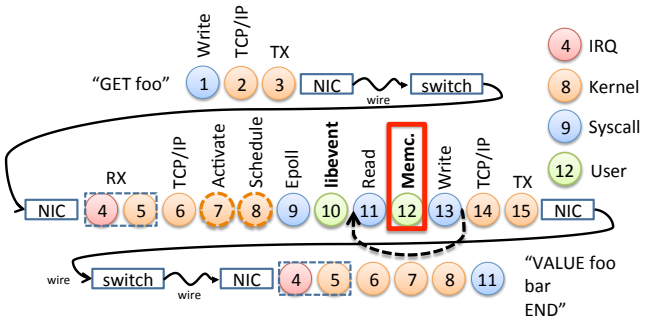

Figure 1. Life of a memcached request. Note that actual request processing (highlighted) is but one step on a long journey.

share hardware resources with the interfering work, by incurring scheduling delay on a subset of threads due to the interfering work, or when the OS migrates threads of the latency-sensitive application on top of each other to make way for the co-located work, causing the application to effectively interfere with itself. A latency-sensitive service's vulnerability to load imbalance can be easily ascertained by purposefully putting it in a situation where threads are unbalanced. For example, we can achieve this by running it with $N + 1$ threads on an N-core system, by forcing only one thread to share a hyper-threaded core with another workload, or by timesharing one core with another workload. By the pigeon-hole principle, such arrangements necessarily reduce the performance of at least one thread of the latency-sensitive service. If the service does not employ any form of load-balancing, this will cause at least one thread of the service to exhibit asymptotic queuing delay at far lower load than the other threads, and is readily observable in measurements of tail latency.

Note that queuing delay, scheduling delay, and imbalance should not be characterized at a fixed load, e.g., at the maximum throughput the server can support or the maximum throughput at which good QoS is still maintained. Since the goal of co-location is to increase utilization when the latency-critical service is at low or medium load, the characterization must be performed across all throughput loads, from 0% to 100%. This provides valuable data across the whole range of likely loads for the latency-critical service. As we show in Sec. 5, it also allows us to work-around interference issues by using cluster-level provisioning instead of server-level optimizations.

3.2 Analysis of Memcached's QoS Sensitivity

3.2.1 Experimental Setup

We now make the preceding analysis concrete by applying it to memcached. For all of the following measurements, we run memcached version 1.4.15 on a dual-socket server populated with Intel Xeon L5640 processors (2× 6 cores @ 2.27 Ghz), 48GB of DDR3-1333 memory, an Intel X520-DA2 10GbE NIC, and running Linux 3.5.0 (with ixgbe driver version 3.17.3). We disable the C6 C-state [34] during our experiments, as it induces high latency (100s of microseconds) at low load due to its wakeup penalty. We also disable DVFS, as it causes variability in our measurements at moderate load. Finally, we disable the `irqbalance` service and manually set the IRQ affinity for each queue of the X520-DA2 to distinct cores (using Intel's `set_irq_affinity.sh` script). Otherwise, memcached performance suffers due to grave softIRQ imbalance across cores. In this configuration, memcached achieves 1.1M queries per second (QPS) for non-pipelined requests and over 3.0M QPS for pipelined requests.

We generate client load for the server with *mutilate*[1], a high-performance, distributed memcached load-generator we developed that can recreate the query distributions at Facebook reported by Atikoglu et al. [1]. Requests are paced using an exponential distribution, and access 200-byte values uniformly at random from a set of 1,000,000 30-byte keys (sized to match the APP pool studied by Atikoglu). Surprisingly, we found no difference in latency with larger datasets or non-uniform access patterns. We do not use variable value sizes in this study, as variance in service time obscures variance due to interference. As memcached is commonly accessed synchronously, we do not pipeline requests. Even in environments that make extensive use of parallel *multi-GETs*, keys are sharded across multiple servers, such that the number of requests handled by any one server is low. In any case, we configure mutilate to make hundreds to thousands of connections to the memcached server, spread across 20 client machines. This ensures that we can generate enough load on the server to saturate it and observe *server-side* queuing delay, while never taxing the clients enough to see *client-side* queuing delay.

3.2.2 Memcached Request Pipeline

We begin the analysis by presenting a detailed description of memcached's request pipeline. There is substantially more to executing a memcached query than just looking up a value in a hash-table. In order to gain a more detailed understanding, we traced the life-cycle of a memcached request in Linux. Fig. 1 depicts the basic steps involved.

A client initiates a request by constructing a query and calling the `write()` system call (1). The request undergoes TCP/IP processing (2), is transmitted by the client's NIC (3), and is then sent to the server's NIC via cables and switches, where upon the processor core running memcached receives an interrupt (since the Intel X520-DA2 NIC maintains flow-to-core affinity with its "Flow Director" hardware [30, 42]). Linux quickly acknowledges the interrupt, constructs a `struct skbuff`, and calls `netif_receive_skb` in *softIRQ* context (4). After determining it is an IP packet, *ip_rcv* is called (5), and after TCP/IP processing is complete,

[1] https://github.com/leverich/mutilate/

Note: all measurements are in microseconds

Who	What	Unl	Ctx Sw	Loaded	L3 int
Server	RX	0.9	0.8	1	1
	TCP/IP	4.7	4.4	4	4
	EPoll	3.9	3.1	2,778	3,780
	libevent	2.4	2.3	3,074	4,545
	Read	2.5	2.1	5	7
	memcached	2.5	2.0	2	4
	Write⋆	4.6	3.9	4	5
	Total	**21.5**	**18.7**	**5,872**	**8,349**
Client	End-to-end	49.8	47.0	6,011	8,460

Table 1. Latency breakdown of an average request when the server process is unloaded (Unl), when it is context-switching with another process (Ctx Sw), when it is fully loaded (Loaded), and when it is subjected to heavy L3 cache interference while fully loaded (L3 int). All measurements are in microseconds. "End-to-end" is the time reported by mutilate on the clients. ⋆For brevity, we include TCP/IP and TX time in Write.

tcp_rcv_established is called (6). At this point, the memcached process responsible for handling this packet has been identified and activated (marked as runnable) (7). Since there is no more packet processing work to be done, the kernel calls `schedule` to resume normal execution (8). Assuming memcached is asleep waiting on an `epoll_wait` system call, it will immediately return and is now aware that there has been activity on a socket (9). If memcached is not asleep at this point, it is still processing requests from the last time that `epoll_wait` returned. Thus, when the server is busy, it can take a while for memcached to even be aware that new requests have arrived. If `epoll_wait` returns a large number of ready file descriptors, it executes them one by one and it may take a long time for memcached to actually call `read` on any particular socket (10). We call this *libevent* time. After returning from `epoll_wait`, it will eventually call `read` on this socket (11), after which memcached finally has a buffer containing the memcached request. After executing the request by looking up the key in its object hash-table, memcached constructs a reply and `write`'s to the socket (13). Now TCP/IP processing is performed (14) and the packet is sent to the NIC (15). The remainder of the request's life-cycle at the client-side plays out similar to how the RX occurred at the server-side. It is interesting to note that a large portion of this request life-cycle is played out in the Linux kernel. We find that only 17% of memcached's runtime is spent in user code vs. 83% in kernel code. Of that 83%, 37% is spent in Soft IRQ context.

Using SystemTap [8], we have instrumented key points in the Linux kernel to estimate how long each step of this process takes. By inspecting the arguments passed to kernel functions and system calls, we are able to create accurate mappings between `skbuffs`, file descriptors, and sockets. Using this information, we can track the latency of *individual requests* as they work their way through the kernel, even though hundreds of requests may be outstanding at any given time. We take measurements for an unloaded case (where only one request is outstanding), a context switching case (where a cpu-bound task is running and the OS must context-switch to memcached after receiving a packet), a loaded case (where memcached is handling requests at peak throughput), and an interference case (where we subject the loaded memcached to heavy L3 cache interference).

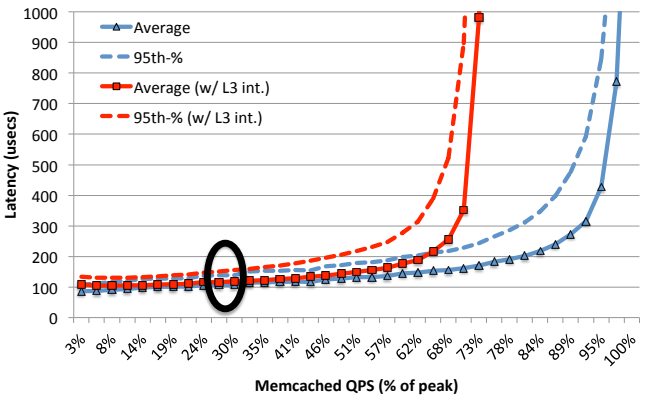

Figure 2. Impact of heavy L3 interference on latency. Interference causes substantial queuing delay at high load, but has little impact at low to moderate load (e.g., at 30%).

3.2.3 Manifestation of Queuing Delay

Table 1 presents the latency breakdown by condensing measurements into key periods: driver RX time, TCP/IP processing, waiting for epoll to return (which includes process scheduling and context switching if memcached isn't already running), libevent queuing delay, read system-call time, memcached execution time, and write system-call time. In the unloaded case there are no surprises: TCP/IP processing, scheduling, and epoll take a plurality of time. We discuss the context-switching case in Sec. 3.2.5. Distinct from the unloaded case, our measurements of the loaded case gives us a key insight: *the vast majority of the latency when memcached is overloaded is queuing delay*. This queuing delay manifests itself in the measurement of "libevent" time, but also "epoll" time. When overloaded, `epoll_wait` is returning hundreds of ready file descriptors. Thus, it will take a while to get to any one request (long "libevent" time). Second, since so many requests are being received by memcached at once, it will take a long time to process them all and call `epoll_wait` again. This shows up in the long "epoll" time measured for subsequent packets. When subjected to interference (for instance, L3 interference), the moderate growth in the time it takes to process each individual request (read, memcached) results in a substantial increase in this queuing delay (epoll, libevent).

This increase in queuing delay due to L3 interference, however, is observed most distinctly at high load. At low

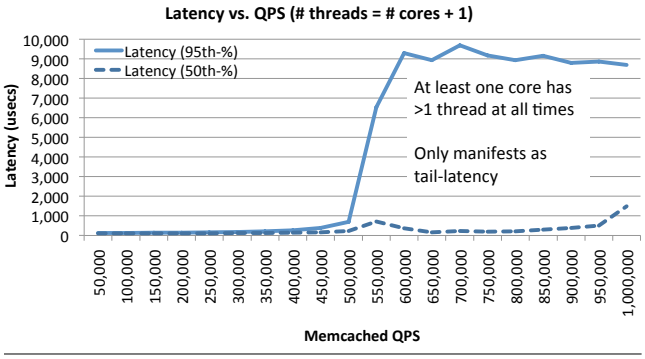

Figure 3. Impact of load imbalance on memcached QoS.

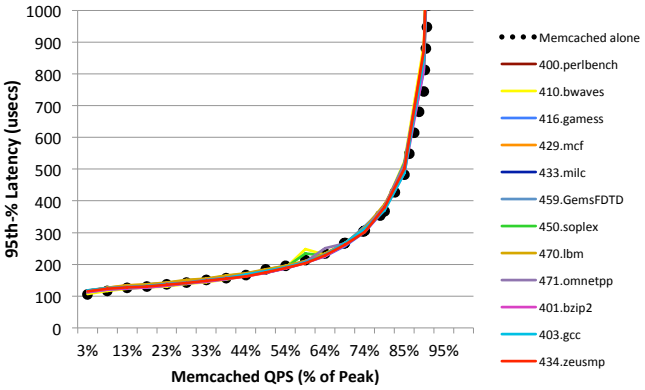

Figure 4. Impact of context-switching with other workloads on memcached latency.

load, the interference is hardly measurable. Fig. 2 plots latency vs. QPS for memcached when subjected to a range of loads. When subjected to L3 cache interference, its service rate is effectively reduced, resulting in asymptotic queuing delay at lower QPS. However, note that whether or not L3 interference is present, latency at 30% QPS is quite similar. This indicates that the impact on per-request latency due to interference is in and of itself insufficient to cause QoS problems at low load. Instead, queuing delay at high load is the real culprit. We build on this observation in Sec. 4.1.

3.2.4 Manifestation of Load Imbalance

As discussed in Sec. 3.1, it is a simple exercise to determine if a service is vulnerable to multi-threaded load imbalance: simply run it with $N+1$ threads on an N-core system. Fig. 3 shows the impact on memcached QPS vs. latency in this scenario (13 threads on a 12-core system). A pronounced spike in tail latency is observed at around 50% of peak QPS not seen in lower-order statistics (e.g., median or average). Interestingly, we found that a similar spike in tail latency is observed even when memcached is run with just 12 threads (instead of 13) on this 12-core system (i.e., N instead of $N+1$ on an N-core system). Upon inspection, we found that Linux frequently migrated threads from one core to another, often ending up with some cores with two threads and other cores with no threads, leading to essentially the same tail latency regressions we observed with 13 threads. Linux's "affine wakeup" mechanism appears to be most responsible for these migrations, causing memcached threads to migrate closer to each other, even when there is only minor contention for shared mutexes. Moreover, Linux's CPU load-balancing mechanism operates at the time scale of 100s of milliseconds, so any mistake in thread placement lasts long enough to be observed in measurements of tail latency. Linux's thread migration issues notwithstanding, these experiments demonstrate that memcached performs no dynamic load balancing across threads. Indeed, inspection of its source code shows that it assigns incoming client connections to threads statically in a round-robin fashion. This lack of load balancing is the root cause of memcached's vulnerability to load imbalance, and a contributing factor to its sensitivity to queuing delay and scheduling delay.

Interestingly, UDP connections to memcached do not suffer from this load imbalance problem in the same way as TCP connections. Memcached monitors the same UDP socket across all threads, and UDP requests are handled by whichever thread reads the socket first. However, kernel lock contention on this UDP socket limits peak UDP throughput at less than 30% of that using TCP. Indeed, Facebook rearchitected memcached's UDP support to use multiple sockets in order to work around this throughput problem [35].

3.2.5 Manifestation of Scheduling Delay

As discussed in Sec. 3.1, scheduling delay consists of context-switch latency and wait time. Seen in Table 1, our SystemTap traces show that not only does memcached not seem to incur any overhead due to the context switch after receiving a request, *it actually goes slightly faster*. We limit the CPU to the C1 C-state for this experiment, so this is not C3 or C6 transition time. We suspect that this overhead has to do with Linux's management of timers for "tickless" idle support, but we have not verified this. In any event, the upshot is that Linux can quickly switch to memcached and service a request with little impact on end-to-end latency, even when another application is running on the CPU core when the request arrives.

Interestingly, contention for the L1 caches or TLBs when timesharing a CPU appears to not be an issue for memcached. Fig. 4 shows latency vs. QPS for memcached when timesharing a CPU with a wide variety of SPEC CPU2006 workloads. In this experiment, we run memcached with "nice -n -20", so that it essentially never incurs wait time; we are only observing the impact of context switching (direct and indirect) with another workload. The fact that memcached is affected so little by contention for the L1 caches and TLBs may be unexpected, but it can be explained intuitively. First, per-request latency is inconsequential compared to the latency incurred from queuing delay, so any in-

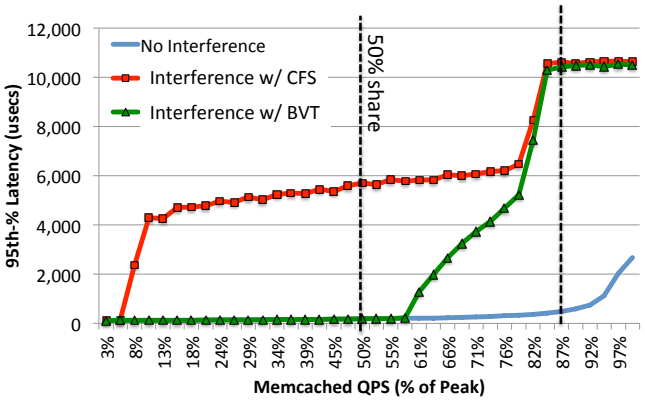

Figure 5. Demonstration of scheduler wait-time induced on memcached when run concurrently with a "square-wave" antagonist. The antagonist runs for 6ms every 48ms (12.5% load). Both memcached and the antagonist are assigned 50% share of the CPU, so memcached should ideally achieve good latency up to that point. However, using the CFS scheduler, memcached exhibits unacceptably high tail latency, even at low load. With BVT (discussed in Sec. 4.3), memcached tail latency is as good as if there were no interference until it exceeds its 50% share of CPU time. Both schedulers cap memcached's throughput at ∼87%, where the CPU is 100% utilized.

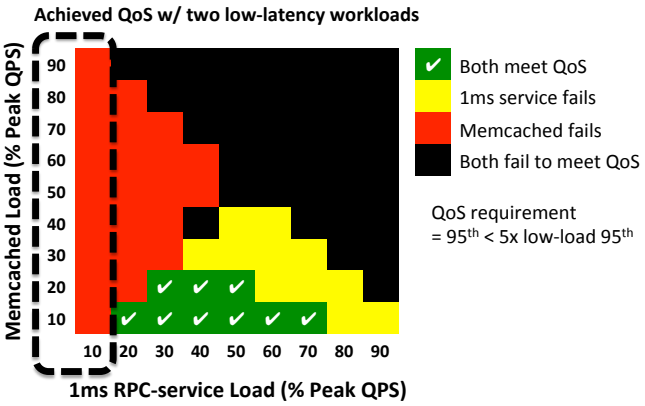

Figure 6. Achieved QoS when co-scheduling two latency-sensitive services on a processor core with Linux's CFS scheduler. For both services, we require that its 95th-% latency never exceed 5× its 95th-% latency as measured at low-load. Both services achieve tolerable QoS only in a narrow range of loads, so it is unwise to co-locate them together without taking corrective action.

crease in per-request latency at low-load is effectively negligible. Second, at high-load, memcached is running often enough to keep its cache footprint warm, so it sees essentially no interference. Finally, since the majority of memcached's runtime is spent in kernel code (83% of runtime), the TLB flushes due to each context switch have little impact; the kernel's code is mapped using large pages. Our conclusion from this experiment is that *context-switching is not itself a large contributor to memcached latency*. If memcached is vulnerable to scheduling delay, it is almost entirely due to wait time induced by the OS scheduling algorithm.

In contrast to context-switch time, memcached is quite vulnerable to scheduler wait time. We demonstrate the scale of the danger by forcing memcached to timeshare a core with a "square-wave" workload. Fig. 5 shows latency vs. QPS for memcached when timesharing a CPU with such a square-wave antagonist, where the antagonist runs in a tight loop (i.e., `while(1);`) for 6ms, sleeps for 42ms, and then repeats; its average CPU usage is only 12.5%. As can be seen, when using CFS (Linux's default scheduler), memcached starts exhibiting exceptionally long tail-latency starting at around 12% of peak QPS, where the aggregate load on the CPU (including both memcached and the antagonist) is only around 25%. The fact that latency spikes when memcached is at approximately the same CPU usage as the antagonist is an interesting artifact of CFS's algorithm; we explore this in detail in Sec. 3.3. At 87% QPS (∼100% CPU load), memcached must yield involuntarily to ensure the antagonist gets fair access to the CPU, so it necessarily experiences long latency.

To illustrate the consequences of this vulnerability in a more realistic setting, we measured QoS for two latency-sensitive services running simultaneously on one server: memcached (with average latency measured on the order of 100us) and a synthetic event-based service similar to memcached with average latency on the order of 1ms. We ran both services concurrently in every combination of QPS from 10% to 100% and monitored latency and throughput. The results are charted in Fig. 6, which reports which of the two services were able to maintain acceptable QoS at each load point. Note that memcached fails to achieve good QoS, even when the co-scheduled workload is offered exceptionally low load (as circled in Fig. 6). Additionally, the 1ms service fails to meet its QoS requirement at higher loads, as it is sensitive to excessive wait time as well. The consequence of this result is that *neither* latency-sensitive service can safely be co-located with the other. Curiously, memcached achieves good QoS when its load is lower than that of the 1ms service; we explain this behavior in Sec. 3.3. Overall, we can conclude that memcached is particularly sensitive to scheduler wait time, and that addressing it (either by adjusting the scheduler or prohibiting CPU time-sharing) is paramount to running memcached with good QoS.

3.3 Scheduling Delay Analysis

We noted in Sec. 3.2.5 that memcached experienced a spike in scheduler wait time right around the point when its CPU usage exceeded that of a co-scheduled application, despite the fact that the CPU is underutilized overall. We describe this behavior in detail in this section.

The essence of the "Completely Fair Scheduler" (CFS) [28], Linux's general-purpose scheduling algorithm, is quite

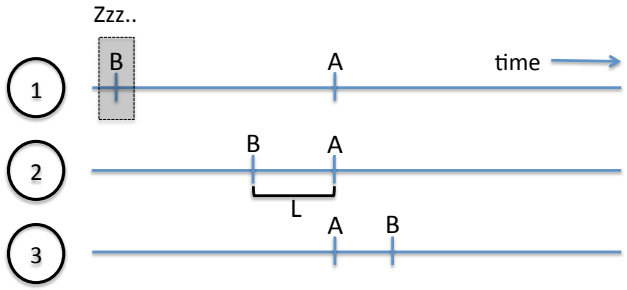

Figure 7. A depiction of CFS run-queue when a task wakes up. Task B is initially asleep and task A is running (1). When task B wakes up, it is placed L behind task A and preempts task A (2), inducing wait time on A while B runs (3).

simple: all non-running tasks are sorted by "virtual runtime" and maintained in a sorted run-queue (implemented as a red-black tree). Tasks accumulate virtual runtime (a weighted measure of actual runtime) when they run. Whenever there is a scheduling event (e.g., a periodic timer tick, an I/O interrupt, a wakeup notification due to inter-process communication, etc.) the scheduler compares the virtual runtime of the current running task with the virtual runtime of the earliest non-running task (if the event is a periodic timer tick) or the task being woken (if the event is a wakeup). If the non-running task has a smaller virtual runtime, the scheduler performs a context switch between the two tasks. Overall, the algorithm attempts to guarantee that the task with the lowest virtual runtime is always the running task, within some error bounded by the maximum period between scheduling events times the number of running tasks. CFS also uses several heuristics to prevent overscheduling (e.g., `sched_min_granularity`). Users may assign "shares" to tasks, and a task's accumulation of virtual runtime when running is weighted inversely proportional to its number of shares. In the long term, tasks receive a fraction of CPU time proportional to the fraction of total shares they are assigned.

An important detail in the design of CFS is how virtual runtime is assigned for a task that wakes up and becomes runnable. The assignment of virtual runtime for waking tasks balances two properties: (1) allowing the waking task to run promptly, in case the event that caused its wakeup needs to be handled urgently, and (2) not giving an unfair share of processor time to waking tasks. CFS balances these two goals by clamping the virtual runtime of a waking task to the minimum virtual runtime of all non-sleeping tasks minus an offset L (called "thresh" in the kernel), which defaults to one half of the target scheduling latency (L = 24ms/2 = 12ms): $\text{vruntime}(T) = \max(\text{vruntime}(T), \min(\text{vruntime}(*)) - L)$.

Unfortunately, CFS's wakeup placement algorithm allows sporadic tasks to induce long wait time on latency-sensitive tasks like memcached. The fundamental problem is that the offset CFS uses when placing the waking task is larger than memcached's nominal request deadline. Illustrated in Fig. 7, if memcached (A) is serving a request when another task B wakes up, it must wait at least for L time before it can resume processing requests. The only way memcached can be guaranteed to never see this delay is if its virtual runtime never exceeds that of the other task. Coming back to Fig. 5, this fully explains why memcached achieves good quality of service when its load is lower than 12%; it is accumulating virtual runtime more slowly than the square-wave workload and always staying behind, so it never gets preempted when the square-wave workload wakes. The same behavior explains why, in Fig. 6, memcached achieves good QoS when its load is less than that of the 1ms service.

3.4 Discussion

The preceding sections have concretely demonstrated how memcached experiences significant reductions in QoS due to queuing delay, scheduling delay, or load imbalance in the presence of co-located workloads. In the course of this study, we additionally evaluated the impact of interference due to network-intensive co-located workloads. Interestingly, these workloads caused far less interference than expected for the following reasons: (1) memcached, particularly with realistic request size distributions, saturates the server's CPUs before it saturates the 10GbE network, and (2) Intel's Flow Director [42] largely isolates co-located workloads from interference due to interrupt handling for large network loads; network interference was indeed a problem when we disabled Flow Director and only used RSS.

It's worth noting that there are a large variety of scenarios where memcached's QoS could suffer even in the absence of co-located workloads. For instance, a misbehaving client could flood the server with requests, denying service to other clients; a client could try to mix large and small requests together, causing long serialization latency for the small requests; the request stream could contain an unusually high number of "SET" requests, which induces frequent lock contention between threads; or the server could genuinely be overloaded from provisioning an insufficient number of servers for a given load. We do not provide a detailed characterization of these vulnerabilities here, as this paper is focused on interference caused by co-located workloads.

4. Addressing QoS Vulnerabilities

This section describes robust strategies that can be employed to address the vulnerabilities identified in Sec. 3 when co-locating latency-sensitive services on servers.

4.1 Tolerating Queuing Delay

In Fig. 2, we demonstrated that interference within the memory hierarchy from co-located workloads only causes tail latency problems when it exacerbates queuing delay. The upshot is that workloads may be safely co-located with each other, despite interference, so long as they don't induce unexpected asymptotic queuing delay. Since queuing delay is a function both of throughput (service rate) and *load*, we can

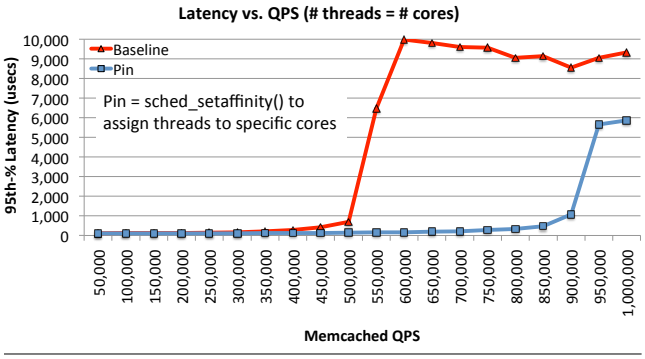

Figure 8. Pinning memcached threads to distinct cores greatly improves load balance, consequently improving tail latency.

tolerate a reduction in throughput (due to interference) if we also reduce the load on the service for any given server. Additional servers can be added to pick up the slack for the low-latency service. Thus, we propose that load be provisioned to services in an *interference-aware* manner, that takes into account the reduction in throughput that a service might experience when deployed on servers with co-located workloads.

Load provisioning is already an important aspect of the life-cycle of distributed datacenter services. Provisioning entails selecting how many and what type of servers are needed to handle a given aggregate request load. While interference-aware provisioning may be pessimistic with respect to how much load an individual server could support for a given service, it accommodates the potential for interference on shared hardware resources and is very permissive in terms allowing other workloads to be co-located on the same hardware. Hence, while interference-aware provisioning may use more servers for the latency-critical service, the TCO for the cluster as a whole may actually be lower because any capacity underutilized by the latency-critical service can be used by other workloads. We give a concrete example of how interference-aware provisioning can improve overall cluster utilization and TCO in Sec. 5.

4.2 Tolerating Load Imbalance

In Sec. 3.2.4, we reported that memcached exhibits a profound spike in latency at around 50% load when running with N threads on an N-core system due to spurious thread-migrations placing multiple threads on a single core. Note that this latency spike is in the absence of co-located work; we cannot operate this server at high load, even if it is dedicated to this workload. One solution to this problem is particularly straight-forward and effective: threads can be pinned explicitly to distinct cores, so that Linux can never migrate them on top of each other. We made a simple 20-line modification to memcached 1.4.15 to query its available CPU set at startup and to statically pin threads (using `sched_setaffinity()`) as they are spawned to each core in round-robin order. After this simple modification, memcached handles up to near 90% of peak load (over 900,000 QPS) with 95th-% latency under 1ms (see Fig. 8). We should also note that the flow-to-core affinity provided by Intel's ixgbe driver for their X520-DA2 NIC also contributes substantially to maintaining balance amongst memcached threads: it ensures that the interrupt load on each core is proportional to its request rate, and that network interrupts destined for co-located workloads do not disturb the cores running memcached [30, 42].

Although thread-pinning is of exceptional utility in this case, it does not address the deeper problem: that memcached makes no attempt to perform load-balancing amongst threads on its own. Thus, it is important to heed the following guidelines when deploying memcached with co-located work. First, it is imperative to pin memcached threads to distinct cores, as demonstrated above. Second, spawn at most N memcached threads on an N-core system; memcached achieves no higher performance with more threads, and tail latency is adversely affected if some cores have more threads than other cores. Third, if you wish to share a CPU core with other workloads (either via OS scheduling or hyper-threading), you might as well share all of the cores that memcached is running on; memcached's tail latency will ultimately be determined by its slowest thread. Finally, if thread load imbalance is unavoidable but certain requests require minimal latency, issue those requests over a UDP connection (Sec. 3.2.4). In the long-term, memcached should be rearchitected to load-balance requests or connections across threads.

4.3 Tolerating Scheduling Delay

As described in Sec. 3.3, the root cause of scheduler-related tail latency lies with CFS's wakeup placement algorithm, which allows workloads which frequently sleep to impose long wait times on other co-located workloads at arbitrary times. Fortunately, there are several strategies one can employ to mitigate this wait time for latency-sensitive services, including (1) adjusting task share values in CFS, (2) utilizing Linux's POSIX real-time scheduling disciplines instead of CFS, or (3) using a general purpose scheduler with support for latency-sensitive tasks, like BVT [7].

4.3.1 Adjusting task shares in CFS

Assigning an extremely large share value to a latency-sensitive task (e.g., by running it with "nice -n -20" or by directly setting its CPU container group's shares value) has the effect of protecting it from wakeup-related wait time. Recall from Sec. 3.3 that a task's virtual runtime advances inversely proportional to the fraction of shares that a particular task possesses relative to all other tasks. Thus, if a task A has a very high shares value, its virtual runtime advances at a crawl and every other task advances at a relative sprint. Thus, each time a task other than A runs, it accrues significant virtual runtime and leaps far ahead of A. Consequently, these tasks are essentially never eligible to preempt

A or induce wait time on it, and effectively only run when A yields.

While effective at mitigating wait-time for latency-sensitive tasks, such a strategy presents a conundrum for servers with co-located workloads: an unexpected spike in load on the latency-sensitive service, or even a bug, could cause its CPU usage to spike and starve the other task. In effect, a task's immunity to wait time and its share of CPU time are tightly coupled in CFS, since the only tunable parameter available to the end-user (shares) affects both. This limitation is fine for some situations (e.g., co-locating best-effort analytics jobs with a memcached cluster at Facebook), but is inappropriate when co-locating multiple user-facing services (e.g., at Google).

4.3.2 POSIX real-time scheduling

An alternative to CFS are the POSIX real-time scheduling disciplines implemented in Linux, `SCHED_FIFO` or `SCHED_RR`.[2] These schedulers are priority-based, where higher-priority tasks may never be preempted by lower-priority tasks, as opposed to the general-purpose fair-share nature of CFS. Tasks scheduled with the POSIX real-time schedulers are implicitly higher-priority than CFS tasks in Linux, so they are never preempted by CFS tasks. Thus, latency-sensitive tasks scheduled using the real-time schedulers never incur CFS-induced wait-time due to other co-located tasks. Additionally, multiple latency sensitive tasks can be safely run concurrently up to an aggregate load of ~70% so long as they are assigned priorities *rate monotonically* (i.e., lower latency begets a higher priority) [20].

Similar to using CFS with a high shares value, the real-time schedulers enable high-priority tasks to starve other tasks; a load-spike or bug in a latency-sensitive service could lead to unfair use of the CPU. Again, this is tolerable when co-located with best-effort workloads, but not with other workloads with throughput or latency requirements.

4.3.3 CPU Bandwidth Limits to Enforce Fairness

For both CFS and the real-time schedulers, Linux allows users to specify a CPU bandwidth limit [39], where a user specifies a runtime "quota" and "period" for a task. This mechanism can prevent high-share tasks from starving others, but it comes with its own drawback: it eliminates the property of "work conservation" from the schedulers. So even if CPU time is unused and available, a task would not be permitted to run if it exceeded its bandwidth quota. While not terribly important at low loads, non-work-conserving means that a task operating at close to its share of CPU time can be hit with long latency as it waits for its quota to be refilled, even if the server is otherwise underutilized.

[2] The principle difference between SCHED_FIFO and SCHED_RR is that SCHED_FIFO tasks run indefinitely until they either voluntarily yield or a higher-priority task becomes runnable, where as SCHED_RR tasks time-share in round-robin order with a slice duration of 100ms.

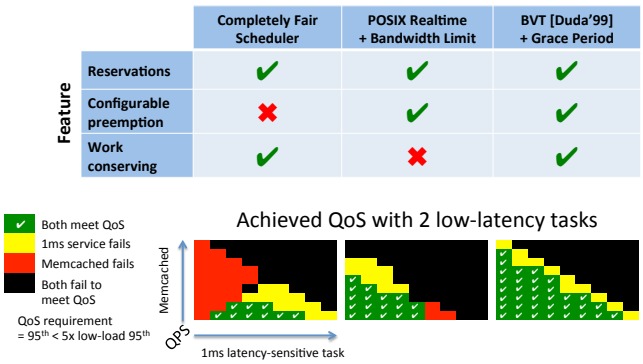

Figure 9. Comparison of scheduling algorithms. "Reservations" indicates whether CPU time can be guaranteed for distinct tasks. "Configuration Preemption" indicates whether the scheduler allows the user to indicate which task may preempt another. See Fig. 9 for axes on the QoS tables.

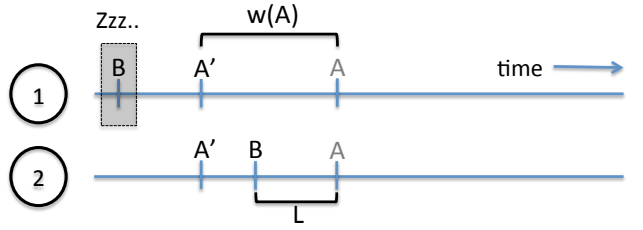

Figure 10. A depiction of BVT run-queue when a task wakes up. Task B is initially asleep. When it wakes up, it gets placed L behind A, as it would with CFS. However, since task A's warp (w(A)) is larger than L, A does not get preempted. Long-term throughput is still determined by the tasks' relative share weights.

We demonstrate the negative consequences of non-work-conservation by repeating the experiment of Fig. 6, where two latency-sensitive tasks are co-located on the same server, but using the real-time SCHED_FIFO scheduler for both tasks instead of CFS. We assigned memcached a higher priority than the 1ms server (i.e., rate-monotonically). We assigned quotas to each of the two services proportional to how high of QPS they were offered at each point (e.g., at 30%/30%, each service is given a quota of 50% of total CPU time), and a period of 24ms (equal to CFS's scheduling period). The result is depicted in the middle column of Fig. 9. In this configuration, the co-location is substantially improved compared to the baseline CFS: there is a large range of moderate loads where both services achieve acceptable QoS. Still, it leaves the server quite underutilized: at least one service degrades when aggregate load exceeds 70%. The degradation is entirely due to the bandwidth limit and non-work-conservation: even though CPU time is available, the services are prohibited from running by fiat.

4.3.4 Borrowed Virtual Time (BVT)

In contrast to CFS and the POSIX real-time schedulers, Borrowed Virtual Time (BVT) [7] affords a more sensible option for operators looking to co-locate latency-sensitive tasks with others. BVT extends virtual-time scheduling by adding a user-provided "warp" parameter that specifies a static offset to be applied to a task's virtual runtime when making scheduling decisions; this adjusted virtual runtime is termed *effective virtual runtime*. This adjustment biases short-term preemption actions, but has minimal impact on long-term throughput fairness, which is still dictated by the fair-share weight assigned to different tasks. This behavior can be seen in Fig. 10. Essentially, BVT allows preemption priority and fair-share allocation to be controlled independently, rather than being tightly coupled as in CFS.

We have implemented BVT as a concise patch to CFS[3] (approx. 150 lines changed) which allows users to assign a "warp" parameter for tasks (or CPU container groups). We additionally added a simple extension to BVT that uses Linux's high-resolution tick infrastructure (HRTICK) to re-evaluate scheduling decisions after a short grace period if a waking task barely misses preempting the current running task; this improves its worst-case scheduling error. Tasks with a warp of 0 behave no differently than regular CFS. The efficacy of BVT is readily apparent when the previous square-wave and latency-sensitive service co-location experiments are rerun using BVT as the scheduler. As seen in Fig. 5, when memcached is run concurrently with the square-wave antagonist using BVT, memcached no longer suffers unacceptable tail latency at low loads. It is not until memcached exceeds 50% load (and starts using more than its fair-share of CPU time) that it begins to incur long latencies due to wait-time. In addition, warp values can be assigned rate-monotonically, much as priorities can be assigned rate-monotonically in POSIX real-time scheduling. In the case of co-locating memcached and a 1ms latency-sensitive service, BVT achieves good QoS for both services up to high aggregate load (80%-90%), as seen in Fig. 9.

Overall, there are several viable strategies to mitigate scheduling delay for latency-sensitive services when co-located with other workloads. When the co-located service is merely "best-effort", it is sufficient to run the latency-sensitive service with high CFS shares. If the co-located service is latency-sensitive or requires a guaranteed portion or fair-share of CPU throughput, a general-purpose scheduler with support for latency-sensitive tasks, like BVT, is required.

5. Co-location Benefits

The preceding sections have shown that a latency-sensitive workload like memcached can be quite resilient to interference caused by co-located workloads. In this section, we analyze the additional utility (performance, power-efficiency, and TCO) that can be extracted from clusters through co-location, either by deploying other workloads onto an existing memcached cluster, or deploying memcached onto an existing batch-compute cluster. We use a combined experimental/analytical approach. We measure latency vs. QPS for memcached with load varying 0% to 100%, while concurrently running SPEC CPU2006 benchmarks on the same server and measuring instruction throughput with Linux's `perf` tool (normalized to the benchmark running alone). We utilize the techniques discussed in Sec. 4 to minimize the interference between the co-located workloads. We use the results to calculate the utility of co-location under two different scenarios explained below.

[3] `https://gist.github.com/leverich/5913713`

	Memcached QPS (% of peak)				
	10%	**30%**	**50%**	**70%**	**90%**
Workload	Memcached 95th-% Latency (usecs)				
none	122	132	156	209	387
400.perlbench	141	161	195	270	474
470.lbm	325	302	309	380	642
mean	217	215	242	307	512
Workload	CPU2006 Instr. per second (norm.)				
400.perlbench	74%	42%	22%	10%	14%
401.bzip2	79%	48%	41%	17%	22%
403.gcc	75%	41%	23%	14%	18%
410.bwaves	65%	29%	17%	9%	10%
416.gamess	78%	49%	28%	17%	17%
429.mcf	77%	54%	38%	28%	26%
433.milc	80%	56%	38%	29%	26%
434.zeusmp	73%	43%	24%	13%	14%
450.soplex	86%	53%	34%	18%	22%
459.GemsFDTD	79%	56%	38%	24%	22%
470.lbm	76%	53%	35%	23%	23%
471.omnetpp	61%	48%	40%	28%	39%
geo. mean	75%	47%	30%	18%	20%
Workload	Perf/TCO-$ improvement (%)				
geo. mean	52%	29%	18%	11%	18%

TCO [29] / server = server + power (3yrs). server = $2,000,
power = $(1 + K_1 + L_1 + K_2 * L_1) * U_{\$,grid} * E$.
$K_1/L_1/K_2 = 1.33/0.8/0.25$, $U_{\$,grid} = \$0.08$/kWh.
E = P * 24 hrs * 365 days * 3 yrs
Average power = P = 0.3 kW * $(0.5 + 0.5*$CPU util.$)$

Table 2. Memcached tail latency and SPEC CPU2006 instruction throughput when co-located under a range of memcached loads. Results are subsetted to save space, but the means consider all benchmarks. Both memcached and SPEC run on all 24 threads of the server and are timesharing. For SPEC CPU throughput, all measurements are of instruction throughput normalized to the benchmark running alone. Perf/TCO-$ improvement compares co-location to operating distinct clusters for the two workloads.

5.1 Facebook Scenario

The first scenario represents the opportunity in companies like Facebook (see discussion in Sec. 2). Memcached is deployed on a large number of dedicated servers that have long periods of underutilization due to diurnal patterns and provisioning for spikes. We'd like to reclaim the unused server resources by also running other workloads (e.g., analytics) as background tasks. Memcached is the high priority workload, and there should be no impact on its QoS.

Table 2 presents the analysis, where the columns represent the memcached service load (as % of peak). The top portion of the table shows the 95th-percentile latency of memcached with and without co-located workloads. While this metric is affected by co-location, it never devolves into asymptotic queuing delay. If the QoS constraint is 1msec, we can co-locate memcached with other workloads at any load. If the QoS limit is 0.5msec, the most memory intensive workloads (470.lbm) can cause memcached to violate its QoS at high load loads (90%). Hence, a conservative approach may be to disallow co-location in the rare case where memcached reaches high levels of utilization.

The table also shows the performance achieved by the SPEC workloads. Since CPU utilization closely matches memcached load, we'd ideally expect a co-located workload to achieve (100-M)% of its peak throughput, where M% is the memcached load. However, SPEC workloads do not achieve ideal performance. 400.perlbench, for instance, only achieves 74.1% throughput when memcached is at 10% load. Curiously, many workloads (e.g., 401.bzip2) achieve slightly higher throughput at 90% memcached load compared to 70% load. We measured substantially fewer context-switches at 90% load even though the request rate was higher, indicating that memcached is servicing more requests per wakeup, and may be the result of interrupt rate throttling by the Intel 10GbE NIC. This results in less time spent scheduling and context-switching and fewer TLB flushes for the co-located workload.

Given the observations above, we calculate the benefit of co-location by comparing the TCO of a single cluster co-locating both of these workloads vs. operating two separate clusters for the two workloads. For the split scenario, we size the cluster for SPEC to match the SPEC throughput provided by the co-located cluster. For instance, if we assume 12,000 servers for memcached and since 400.perlbench achieves 74.1% throughput when co-located with memcached at 10% load, this throughput can be matched by 8,892 servers dedicated to running 400.perlbench. For TCO, we use the methodology of Patel et al. [19, 29] to compute the fully-burdened cost of power using the parameters in Table 2 and the assumptions that (1) server capital cost is $2,000, (2) server power consumption scales linearly with CPU utilization, and static power is 50% of peak power, (3) peak power consumption is 300W, and (4) datacenter PUE is 1.25. This methodology takes into account the fact that a server operating at higher utilization consumes more power, hence it has higher overall cost.

	Memcached QPS (% of peak)				
	50%	**60%**	**70%**	**80%**	**90%**
Workload	Memcached 95th-% Latency (usecs)				
none	156	176	210	250	360
400.perlbench	156	177	206	266	380
401.bzip2	156	181	213	268	402
403.gcc	162	183	217	282	457
429.mcf	186	228	318	4,246	6,775
459.GemsFDTD	201	253	408	6,335	6,896
470.lbm	208	274	449	7,799	6,429
mix	157	199	207	334	4,711
L3 μbench	156	197	246	5,400	4,944
Workload	CPU2006 Instr. per second (norm.)				
400.perlbench	99%	98%	98%	98%	98%
470.lbm	91%	90%	89%	88%	89%
mix	98%	98%	98%	96%	94%
geo. mean	98%	97%	96%	97%	97%

Table 3. Memcached tail latency and SPEC CPU2006 instruction throughput when co-located on distinct cores. Results are subsetted to save space, but the mean considers all benchmarks. Each workload gets half of the cores of the server. Memcached peak QPS is measured when running alone on its half of the cores. Shaded cells indicate an unacceptable drop in QoS.

Despite the fact that SPEC workloads do not achieve ideal performance when co-located with memcached, the TCO benefit of co-location in all cases is positive and substantial. When memcached operates at 10% load, the co-located workloads achieve an average 75.4% of the throughput of an equivalent-sized cluster (12,000 servers). To match this performance with a separate cluster, it would cost an additional 52% increase in TCO, taking into account all capital and operational expenses. To put it differently, it is as-if we achieve the capability of 9,048 servers running SPEC at a 34% discount (1-$\frac{1}{.52+1}$). Even at 90% load for memcached, there is still a 18% TCO benefit to co-locating workloads.

5.2 Google Scenario

In the second scenario, we assume an underutilized cluster similar to the 12,000-server cluster analyzed by Reiss et al. [32] (20% CPU utilization, 40% memory utilization). We conservatively assume that half (50%) of the CPU cores in the cluster are utilized running some primary workload and we would like to evaluate the benefit of deploying memcached on the other available cores in order to take advantage of the $\sim$0.5 PB of unallocated memory in the cluster.

Again, we evaluate the efficacy of co-location by comparing a co-located cluster to a pair of separate clusters where one cluster runs the existing primary workload (SPEC workloads at 50% core utilization) and the other cluster runs

memcached. The impact on latency and throughput of co-location at various memcached loads is presented in Table 3. For many workloads (perlbench, bzip2), the latency impact of co-location is negligible. For others (lbm, mcf, etc.), substantial queuing delay is observed at loads above 70%, as previously seen in Sec. 3.2.3. Moreover, the SPEC workloads in some cases also see a moderate slowdown (up to 12% for lbm). If we assume that 500 μsec 95th-% latency is the maximum latency we are willing to tolerate from memcached, and 10% is the maximum slowdown we are willing to tolerate from the SPEC workloads, then the maximum memcached load we can provision for co-located servers is 60% of peak (*interference-aware provisioning*).

Were we to build a separate memcached cluster to accommodate this load, we would allow memcached to use all of the cores of those servers. Thus, we would need to increase the size of the cluster by an additional 30%, optimistically assuming that performance scales linearly with cores. Put another way, if our original cluster is 12,000 servers large and 50% is occupied with SPEC workloads, we can safely co-locate up 3,600 servers worth of additional memcached load to serve unallocated memory, and guarantee good quality of service for both workloads. Taking into account TCO and the pessimistic power assumption that the memcached service is always at 100% load, the co-located cluster achieves 17% improvement in TCO compared to two separate clusters with the same performance for the two workloads. Also note that the Google cluster studied by Reiss et al. had 60% of memory unallocated; a separate memcached cluster of this *capacity* would require 7,200 servers at substantially higher TCO.

6. Related Work

There have been several works addressing QoS for co-located workloads in warehouse-scale datacenters [6, 22, 23, 43–45]. Several of these, including Paragon, Bubble-Up, and Bubble-Flux, focus on identifying workloads which interfere with each other and avoiding co-locating them together. Our work is distinct in that we (1) concretely describe how this interference manifests in a canonical low-latency workload, and (2) accommodate co-locations that may cause substantial interference by considering the overall impact on load provisioning in our analytical study. Other works focus on minimizing latency or improving tail latency for specific low-latency services [17, 27]. They accomplish this by bypassing the OS and using user-level network stacks, which complicates workload consolidation; these services would invariably have low utilization during diurnal troughs, which is counter-productive for the goals of this work. Tessellation [3] and Akaros [33] improve QoS in operating systems through radical changes to the kernel and process abstractions. Our work instead delves deep into the challenge of co-locating workloads using existing OS abstractions, and presents specific examples of where the epoll interface and CFS fall short in Linux. Pisces [38] presents a holistic solution to QoS for membase; however, they do not consider workload co-location, just multi-tenancy for membase. Finally, hardware partitioning techniques have been studied for caches [31, 36], hyper-threads [12], and memory channels [25, 26]. Those works are orthogonal to our own.

7. Conclusions

In this paper, we address the conflict between co-locating workloads to increase the utilization of servers and the challenge of maintaining good quality-of-service for latency-sensitive services. We showed that workload co-location leads to QoS violations due to increases in queuing delay, scheduling delay, and thread load imbalance. We also demonstrated concretely how and why these vulnerabilities manifest in a canonical low-latency service, memcached, and described several strategies to mitigate the interference due to them. Ultimately, we demonstrated that latency-critical workloads like memcached can be aggressively co-located with other workloads, achieve good QoS, and that such co-location can improve a datacenter's effective throughput per TCO-$ by up to 52%.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. SIGMETRICS, 2012.

[2] Luiz Andre Barroso. Warehouse-Scale Computing: Entering the Teenage Decade. ISCA, 2011.

[3] Juan A. Colmenares et al. Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation. DAC, 2013.

[4] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commununications of the ACM*, 56(2):74–80, February 2013.

[5] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. IISWC, 2013.

[6] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. ASPLOS, 2013.

[7] Kenneth J Duda and David R Cheriton. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler. SOSP, 1999.

[8] Frank C. Eigler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of Systemtap: A Linux Trace/Probe Tool, 2005.

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. ISCA, 2011.

[10] Gartner says efficient data center design can lead to 300 percent capacity growth in 60 percent less space. `http://www.gartner.com/newsroom/id/1472714`, 2010.

[11] Donald Gross, John F Shortle, James M Thompson, and Carl M Harris. *Fundamentals of Queueing Theory*. Wiley, 2013.

[12] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten, and Martin Dixon. SMT QoS: Hardware Prototyping of Thread-level Performance Differentiation Mechanisms. HotPar, 2012.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. NSDI, 2011.

[14] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[15] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaj i Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. NSDI, 2013.

[16] James M Kaplan, William Forrest, and Noah Kindler. Revolutionizing Data Center Energy Efficiency. Technical report, McKinsey & Company, 2008.

[17] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Ãmin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. SOCC, 2012.

[18] David G Kendall. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, 1953.

[19] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. ISCA, 2008.

[20] Chung Laung Liu and James W Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[21] Huan Liu. A Measurement Study of Server Utilization in Public Clouds. In *Proc. of the Intl. Conference on Dependable, Autonomic and Secure Computing*, 2011.

[22] Jason Mars, Lingjia Tang, and Robert Hundt. Heterogeneity in "Homogeneous" Warehouse-Scale Computers: A Performance Opportunity. *IEEE Computer Architecture Letters*, 10(2), 2011.

[23] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proc. of the Intl. Symposium on Microarchitecture*, 2011.

[24] David Meisner, Junjie Wu, and Thomas F Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. ISPASS, 2012.

[25] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proc. of the Intl. Symposium on Microarchitecture*, 2007.

[26] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair Queuing Memory Systems. In *Proc. of the Intl. Symposium on Microarchitecture*, 2006.

[27] John Ousterhout et al. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4), 2010.

[28] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[29] Chandrakant D. Patel and Amip J. Shah. Cost Model for Planning, Development and Operation of a Data Center. Technical report HPL-2005-107R1, Hewlett-Packard Labs, 2005.

[30] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T Morris. Improving Network Connection Locality on Multicore Systems. EuroSys, 2012.

[31] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of the Intl. Symposium on Microarchitecture*, 2006.

[32] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. SOCC, 2012.

[33] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. SOCC, 2011.

[34] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-Management Architecture of the Intel Microarchitecture Code-named Sandy Bridge. *IEEE Micro*, 32(2), 2012.

[35] Paul Saab. Scaling memcached at Facebook. `https://www.facebook.com/note.php?note_id=39391378919`, December 2008.

[36] Daniel Sanchez and Christos Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. *IEEE Micro's Top Picks*, 32(3), May-June 2012.

[37] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. EuroSys, 2013.

[38] David Shue, Michael J Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. OSDI, 2012.

[39] Paul Turner, Bharata B Rao, and Nikhil Rao. CPU Bandwidth Control for CFS. Linux Symposium, 2010.

[40] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T Sivabalan, and Rajesh Subbiah. Worth Their Watts?–An Empirical Study of Datacenter Servers. HPCA, 2010.

[41] VMware. VMware Infrastructure: Resource Management with VMware DRS. White paper, VMware, 2006.

[42] Wenji Wu, Phil DeMar, and Matt Crawford. Why Can Some Advanced Ethernet NICs Cause Packet Reordering? *IEEE Communications Letters*, 15(2):253–255, 2011.

[43] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. NSDI, 2013.

[44] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. ISCA, 2013.

[45] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU Performance Isolation for Shared Compute Clusters. EuroSys, 2013.